A SECD Machine for PCF

Bradley C. Kuszmaul

December 17, 1986

A Term Paper for Logic and Semantics of Programs

(MIT Course 6.830J/18.427J)

Fall, 1986

Abstract: The SECD machine approach to providing an operational semantics for call-by-value applicative languages is well established in the literature. Our goal is to provide a similar operational semantics for a *call by name* applicative language called PCF. The definition of PCF available in the literature is given by another operational semantics, and we show that our SECD operational semantics agree with the given operational semantics. We actually define two different SECD machines for PCF and compare their merits.

Introduction

The language PCF is defined in [Plo77] and an operational semantics is given. We will review those semantics later in the paper. One important feature of the semantics of PCF is that PCF has *call by value* semantics. This means that the argument to a function is not evaluated unless that argument is actually used by the function. Note that since PCF is an purely functional applicative language, the semantics do not specify exactly *when* something is evaluated, but they can specify *whether* something is evaluated. In particular, if some interpreter for PCF could *prove* that a particular function actually used its argument, that interpreter could evaluate the argument to the function before interpreting the body of the function. Correspondingly, if the interpreter could prove that the evaluation of the argument to a function was always well defined (i.e. the evaluation terminated), then the interpretor could arrange to evaluate the argument before looking at the body of the function, and the semantics would be preserved. Call by value semantics merely requires that if an argument to a function is never used, and the evaluation of the argument is not well defined, then the interpretor must not get stuck trying to evaluate the argument.

One approach to defining the operational semantics of a programming language is called the *SECD machine*. A SECD machine is an automaton which interprets a language. (The acronym stands for the initials of the register names in the automaton, and the name survives for the same sort of reasons that the names CAR and CDR survive in LISP.) The "original" SECD machine given in [Lan63] is a machine to interpret a language with *call by value* semantics. For such a language, the arguments to functions are always evaluated. This means that even if the argument is never used, the result of evaluating the application of a function to a "runaway" argument is undefined. Many programming languages in use today use call by value. We will show a SECD machine for PCF, which means that we will have to modify the SECD machine idea to implement call by name instead of call by value.

Another approach to defining an operational semantics is to provide a set of *rewrite rules* for the language. The rewrite rules show how to transform a given expression into some other

expression which is hopefully closer to the final answer. In particular after enough rewrites, we hope that the expression has been translated into a numeral which we read as an integer. Much of grade school algebra consists of these sorts of rewrite rules. An example of such a rewrite rule for grade school algebra is:

$$(k_i + k_n) \rightarrow k_{i+n}$$

where k_i is the numeral which represents the integer i.

The definition of PCF which can be found in [Plo77] has an operational semantics based on rewrite rules. The particular rewrite rules given in [Plo77] were shown to have the property that at most one of the rewrite rules can be applied at any time. This means that the "rewrite operator" (which we write as $\rightarrow_{\text{pcf}}$) is a mathematical *partial function* from expressions in PCF to expressions in PCF. (Some expressions in PCF can not be rewritten as anything under those rewrite rules in which case we think of the computation as having completed.) The definition of EVAL on an expression is to apply the $\rightarrow_{\text{pcf}}$ operator as many times as possible until the expression can not be rewritten any more. For some expressions, the rewrites go on forever, and EVAL is undefined on such expressions.

In this paper, we first review the definition of PCF. We will show two SECD machines for evaluating PCF expressions. The first SECD machine will look a lot like the rewrite rules for PCF and will use fairly complex primitive operations (in particular substitution of an expression for a variable in another expression is a primitive for our first SECD machine). We will then show a second SECD machine which uses primitive operations which are less complex. The tradeoff is that it is a little harder to show that the second SECD machine agrees with the rewrite rules for PCF. We will also compare the relative performance of the two machines when implemented in a typical programming language such as LISP.

The Programming Language, PCF

In this paper, we assume that the reader is familiar with the first few pages of [Plo77], (in particular the definition of PCF). We will ignore type information in this paper, since if one is given a PCF program without type annotations, and if there is some type annotation for the PCF program, then that type annotation is unique. We will assume that all the programs we deal with are in fact type correct in the sense that there is a "typing" for the program, and that the reader understands how to give an arbitrary program type information by a type inference algorithm.

As a concession to making this paper self contained, we include a brief restatement of the constants and the rewrite rules for PCF.

We consider the language $\mathcal{L}_A$ for arithmetic with the following constants:

tt: the true boolean,

ff: the false boolean,

$\supset$: the "conditional" or "IF-THEN-ELSE" operator,

Y: the fixed point operator,

k_n **(for $n \in \mathbf{N}$)**: the nth numeral (i.e. the numeral which denotes integer n),

$(+1)$: the integer increment operator,

(-1): the integer decrement operator, and

Z: the zero test (which translates integers into booleans).

The operational semantics of PCF is given by a partial function, EVAL, which gives constants from programs. The EVAL function is defined by means of an *immediate reduction relation* (which we are calling a *rewrite rule*), $\rightarrow_{\text{pcf}}$ between terms by:

$$\text{EVAL}(M) = c \text{ iff } M \rightarrow_{\text{pcf}}{}^{*} c,$$

where M is a program and c is a constant.

The definition of $\rightarrow_{\text{pcf}}$ is given by some rules:

(I1): $(\supset\ tt\, M\, N) \rightarrow_{\text{pcf}} M,\ (\supset\ ff\, M\, N) \rightarrow_{\text{pcf}} N$

(I2): $(Y\, M) \rightarrow_{\text{pcf}} (M(Y\, M))$,

(I3): $((\lambda\alpha M)N) \rightarrow_{\text{pcf}} [N/\alpha]M$,

(I4): $((+1)k_m) \rightarrow_{\text{pcf}} k_{m+1}$ (for $m \geq 0$),

(I5): $((-1)k_{m+1}) \rightarrow_{\text{pcf}} k_m$ (for $m \geq 0$),

(I6): $(Z\, K_0) \rightarrow_{\text{pcf}} tt,\ (Z k_{m+1}) \rightarrow_{\text{pcf}} ff$,

(II1): $$\frac{M \rightarrow_{\text{pcf}} M'}{(M\, N) \rightarrow_{\text{pcf}} (M'\, N)}$$

(II2): $$\frac{M \rightarrow_{\text{pcf}} M'}{(\supset\ M) \rightarrow_{\text{pcf}} (\supset\ M')}$$

(II3): $$\frac{N \rightarrow_{\text{pcf}} N'}{(M\, N) \rightarrow_{\text{pcf}} (M\, N')}$$ (if M is $(+1)$, (-1), or Z).

The relation $\rightarrow_{\text{pcf}}$ is actually a partial function, which is undefined on constants and so EVAL is well defined as a partial function. (Note that EVAL is undefined on those expressions which "run forever".)

Note that rule I3 uses the notation $[N/\alpha]M$. This means to substitute N for all free occurences of α in the expression M. To do that may require changing the bound variables of M so that no free variables of M become bound. We could define $[N/\alpha]M$ by a recursive definition as follows:

$$\begin{aligned} [N/\alpha]c &= c \text{ for } c \text{ a constant}, \\ [N/\alpha](M_1 M_2) &= ([N/\alpha]M_1\ [N/\alpha]M_2), \\ [N/\alpha](\lambda\beta M) &= (\lambda\gamma[N/\alpha]([\gamma/\beta]M)) \quad \text{where } \gamma \text{ is a variable which does not occur in } M \text{ or } N. \end{aligned}$$

We can make this definition be well defined by always choosing γ to be the first variable (in the canonical enumeration of all variables) which does not occur in either M or N. If we

were writing a computer program to implement this substitution operator we might be more careful and not change things unless they actually had to be changed. For example if β is not free in N then we could have defined $[N/\alpha](\lambda\beta M)$ to be $(\lambda\beta[N/\alpha]M)$.

Our First SECD Machine for PCF

The problem with the rewrite rules given by $\rightarrow_{\text{pcf}}$ is that they do not give a good idea about how an automaton could go about reducing a PCF program to a constant. The goal of a SECD machine is to change the rewrite rules so that each state change is "local". Our first SECD machine comes closer to meeting this goal by getting rid of the recursive definitions that are used in the definition of $\rightarrow_{\text{pcf}}$, but it still uses the variable substition operator as a primitive operation. As we saw in the last section the variable substition operator is itself fairly complex. One could imagine building a SECD-like machine just to do variable substitution, and merging it into our SECD machine. Later (in the next section), we will define a different SECD machine which does not do variable substitution in the same way at all.

Our first SECD machine uses two registers to hold its state: The V (value) register, which holds an expression which is being evaluated, and the D (dump) register, which holds information about how to use that value when there are no more "rewrites" to apply to it. The transformations of our SECD machine will correspond closely to the rewrite rules given above. To make this work, we have chosen to make substitution of an expression for a variable in another expression a primitive operation for our SECD machine.

The D register on our machine can be thought of as a stack. It is possible to "push" things onto the D register and to look at the top few things on the D register. In particular, when rule II1, II2, or II3, is applicable as a rewrite rule to the expression in the V register, the rewrite rule calls for rewriting a subexpression and replacing it in the original expression. That is too much for our little machine to do all in one step, so the machine pushes enough information onto the D register in order to get back to its original state.

To get an idea of what we need to remember on the D stack, consider each rule II1, II2, and II3:

Thus for rule II1, which states that if $M \rightarrow_{\text{pcf}} M'$ then $(MN) \rightarrow_{\text{pcf}} (M'N)$, we need to save N plus the fact that we were on rule II1, and then put M onto the V register for the machine to evaluate. We save the fact that we were on rule II1 by pushing a special pair $\langle \mathbf{arg}, N \rangle$ onto the D stack. If the machine gets stuck evaluating something, we look on the D register, and if we see $\langle \mathbf{arg}, M \rangle$ as the top element of the D stack, then we can try to apply the V value to the M value. We will be more specific on how to perform this application later. We can think of rule II1 as corresponding to pushing an unevaluated argument onto the D stack.

For rule II2, which states that if $M_o \rightarrow_{\text{pcf}} M_0'$ then $(\supset_\sigma\ M_0) \rightarrow_{\text{pcf}} (\subset_\sigma\ M_0'$, we merely need to remember that we were on rule II2, and put M_o on the V register. We do this by pushing a special token, **if**, onto the D stack. If we get stuck evaluating V and notice **if** on top of the D stack, then we check to make sure that V is *tt* or *ff*, and that there are enough "arguments" on the D stack to try to apply rule I1.

For rule II3, which states that if $N \rightarrow_{\text{pcf}} N'$ and M is $+1$, -1, or Z, then $(MN) \rightarrow_{\text{pcf}} (MN')$, then we need to remember M, and the fact that we are working on rule II3 onto the stack. We do this by pushing the pair $\langle \mathbf{op}, M \rangle$ onto the D stack.

Now, that is not exactly how our SECD machine works. The actual rules for our SECD machine are as follows:

We write the state of our SECD machine as an ordered pair, $\langle V, D \rangle$.

We will write the D stack as a list $(a, b, c, d, \ldots)$, which means that a is on the top of the stack, b is the second element of the stack, and so on. We sometimes use a LISP-like "DOT" notation for lists, so for example $(a, b\ .\ c)$ is the list with first element a, second element b, and the "rest" of the stack is c. Another example is that $(a, b, \ .\ (cd))$ is the same as (a, b, c, d).

Our operational semantics are going to assume that the original program is type correct, and we are going to ignore the type checking issues. Most readers should be able to annotate

our SECD machine with type information.

Here is the definition of $\rightarrow_a$, which maps SECD_a machine state pairs into state pairs:

(I1-a) $\langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1\rangle, \langle \mathbf{arg}, M_2\rangle \,.\, D')\rangle \rightarrow_a \langle M_1, D'\rangle$,

(I1-b) $\langle ff, (\mathbf{if}, \langle \mathbf{arg}, M_1\rangle, \langle \mathbf{arg}, M_2\rangle \,.\, D')\rangle \rightarrow_a \langle M_2, D'\rangle$,

(I2) $\langle Y, (\langle \mathbf{arg}, M\rangle \,.\, D')\rangle \rightarrow_a \langle (M(YM)), D'\rangle$,

(I3) $\langle (\lambda\alpha M), (\langle \mathbf{arg}, N\rangle \,.\, D')\rangle \rightarrow_a \langle [N/\alpha]M, D'\rangle$,

(I4) $\langle k_m, (\langle \mathbf{op}, +1\rangle \,.\, D')\rangle \rightarrow_a \langle k_{m+1}, D'\rangle$ (for $m \geq 0$),

(I5) $\langle k_{m+1}, (\langle \mathbf{op}, -1\rangle \,.\, D')\rangle \rightarrow_a \langle k_m, D'\rangle$ (for $m \geq 0$),

(I6-a) $\langle k_0, (\langle \mathbf{op}, Z\rangle \,.\, D')\rangle \rightarrow_a \langle tt, D'\rangle$,

(I6-b) $\langle k_{m+1}, (\langle \mathbf{op}, Z\rangle \,.\, D')\rangle \rightarrow_a \langle ff, D'\rangle$ (for $m \geq 0$),

(II1) $\langle (MN), D\rangle \rightarrow_a \langle M, (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle$,

(II2) $\langle \supset, (\langle \mathbf{arg}, M\rangle \,.\, D')\rangle \rightarrow_a \langle M, (\mathbf{if} \,.\, D')\rangle$,

(II3) $\langle M, (\langle \mathbf{arg}, N\rangle \,.\, D')\rangle \rightarrow_a \langle N, (\langle \mathbf{op}, M\rangle \,.\, D')\rangle$ (if M is $(+1)$, (-1), or Z).

We observe that given any state $\langle V, D\rangle$, there is at most one $\rightarrow_a$ rule that applies, so in fact the $\rightarrow_a$ relation is a partial function for state pairs to state pairs.

Define the function SECD_a from PCF programs to PCF constants to be

$$\text{SECD}_a(M) = c \text{ iff } \langle M, ()\rangle \rightarrow_a^* \langle c, ()\rangle,$$

where c is a constant in PCF, and M is a program in PCF (and of course $()$ is the empty list).

Here $\rightarrow_a^*$ is the transitive reflexive closure of $\rightarrow_a$. For this definition to make sense it is necessary that if $\langle M, ()\rangle \rightarrow_a^* \langle c, ()\rangle$ and $\langle M, ()\rangle \rightarrow_a^* \langle c', ()\rangle$ then c and c' are identical, but this follows from the easy observation that the $\rightarrow_a$ relation is really a function, so $\langle M, ()\rangle$ must be mapped into some well defined sequence of intermediate states by $\rightarrow_a^i$, and since there is no $\langle V, D\rangle$ such that $\langle c, ()\rangle \rightarrow_a \langle V, E\rangle$, then $\langle c, ()\rangle$ is the "end of the line" and it follows that $c = c'$.

In a sense SECD_a is the function which applies the $\rightarrow_a$ rule to an expression and an empty D register over and over until it can not any more. Note that for pairs $\langle V, ()\rangle$ which "run

forever", the value of $\mathrm{SECD}_a(V)$ is undefined.

We need a proof that this two register SECD machine correctly interprets PCF. In particular we will show that it agrees with the rewrite rules for PCF as stated by the following theorem:

Theorem 1 *Given any expression E of ground type,* $\mathrm{EVAL}(E)$ *is well defined if and only if* $\mathrm{SECD}_a(E)$ *is well defined, and if they are well defined, then* $\mathrm{EVAL}(E) = \mathrm{SECD}_a(E)$.

Our proof is by induction on the structure of the expression to be evaluated (i.e. *structural induction*). We prove some property for all the constants in PCF, and then we assume that if the property holds for M and N we show that it holds for $(M\ N)$ and $(\lambda\alpha\ M)$.

Lemma 1 *If $V \rightarrow_{pcf} V'$ then for all stack values D, $\exists n, m, V'', D''$ where n and m are non-negative integers, V'' is a PCF expression, and D'' is a stack value, such that*

$$\begin{aligned} \langle V, D\rangle &\rightarrow_a^n \langle V'', D''\rangle, \text{ and} \\ \langle V', D\rangle &\rightarrow_a^m \langle V'', D''\rangle. \end{aligned}$$

Since we are assuming the results of[Plo77], we know that for any PCF expression E of ground type, $\mathrm{EVAL}(E)$ is a constant (of that ground type) if and only if $\mathrm{EVAL}(E)$ is well defined. (This follows since if $\mathrm{EVAL}(E)$ is well defined and is not a constant then by Theorem 3.1 of[Plo77] the semantical meaning of E is not denoted by a constant which means that the semantical meaning of E must be $\perp$ but then $\mathrm{EVAL}(E)$ is not well defined which is a contradiction.)

This means that Lemma 1 proves Theorem 1 for all E of ground type since if $E \rightarrow_{\mathrm{pcf}} c$ for some constant c (of ground type), then letting $V = E$, $V' = c$, and $D = ()$ (the empty list) we have some n, m, V'', D'' such that

$$\langle E, ()\rangle \rightarrow_a^n \langle V'', D''\rangle, \text{ and}$$

$$\langle c, ()\rangle \rightarrow_a^m \langle V'', D''\rangle.$$

But note that $\langle c, () \rangle$ does not transform to anything under $\rightarrow_a$, so m must be zero, and so we must have that $V'' = c$, and $D'' = ()$, so we know that $\langle E, () \rangle \rightarrow_a^n \langle c, () \rangle$.

Now for the proof of Lemma 1: This proof uses structural induction on V.

Suppose we have that $V \rightarrow_{\text{pcf}} V'$. This means that one of the rewrite rules for PCF applies. We do a case analysis: Suppose the rewrite rule that transforms V to V' is

rule I1: Then V is of the form $(\supset\ tt\, M_1\ M_2)$ or $(\supset\ tt\, M_1\ M_2)$.

Suppose $V = (\supset\ tt\, M_1\ M_2)$: Then since $V \rightarrow_{\text{pcf}} V'$ we have that $V' = M_1$. First we need to curry the expression so that it makes sense in PCF:

$$\begin{aligned} V &= (\supset\ tt\, M_1\ M_2) \\ &= (((\supset\ tt) M_1) M_2) \end{aligned}$$

Note that for any D we have

$$\begin{aligned} \langle (((\supset\ tt) M_1) M_2), D \rangle &\rightarrow_a \langle ((\supset\ tt) M_1), (\langle \mathbf{arg}, M_2 \rangle\ .\ D) \rangle \text{ (by SECD}_a \text{ rule II1)} \\ &\rightarrow_a \langle (\supset\ tt), (\langle \mathbf{arg}, M_1 \rangle, \langle \mathbf{arg}, M_2 \rangle\ .\ D) \rangle \text{ (by SECDa rule II1)} \\ &\rightarrow_a \langle \supset, (\langle \mathbf{arg}, tt \rangle, \langle \mathbf{arg}, M_1 \rangle, \langle \mathbf{arg}, M_2 \rangle\ .\ D) \rangle \text{ (by SECD}_a \text{ rule II1)} \\ &\rightarrow_a \langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1 \rangle, \langle \mathbf{arg}, M_2 \rangle\ .\ D) \rangle \text{ (by SECD}_a \text{ rule II2)} \\ &\rightarrow_a \langle M_1, D \rangle \text{ (by SECD}_a \text{ rule I1-a).} \end{aligned}$$

So letting $n = 5$, $m = 0$, $V'' = V'$, $D'' = D$ does the job.

Suppose $V = (\supset\ ff\, M_1\ M_2)$: This is similar to the previous case except that at the last step instead of applying applying rule I1-a, we apply rule I1-b.

rule I2: Then $V = (Y\, M)$ for some M and $V' = (M\, (Y\, M))$. Note that for any D we have

$$\begin{aligned} \langle (Y\, M), D \rangle &\rightarrow_a \langle Y, (\langle \mathbf{arg}, M \rangle\ .\ D) \rangle \text{ (by SECD}_a \text{ rule II1)} \\ &\rightarrow_a \langle (M\, (Y\, M)), D \rangle \text{ (by SECD}_a \text{ rule II2).} \end{aligned}$$

Thus $n = 2$, $m = 0$, $V'' = V'$, $D'' = D$ does the job.

rule I3: Then $V = ((\lambda\alpha M)N)$ for some M and N, and $V' = [N/\alpha]M$. Thus for any D we have

$$\begin{aligned}\langle((\lambda\alpha M)N), D\rangle &\rightarrow_a \langle(\lambda\alpha M), (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle \text{ (by SECD}_a \text{ rule II2)}\\ &\rightarrow_a \langle[N/\alpha]M, D\rangle \text{ (by SECD}_a \text{ rule I3).}\end{aligned}$$

Thus $n = 2$, $m = 0$, $V'' = V'$, $D'' = D$ does the job.

rule I4: Then $V = (+1)k_m$ for some $m \geq 0$ and $V' = k_{m+1}$. Then for any D we have

$$\begin{aligned}\langle(+1)k_m, D\rangle &\rightarrow_a \langle+1, (\langle \mathbf{arg}, k_m\rangle \,.\, D)\rangle\\ &\rightarrow_a \langle k_m, (\langle \mathbf{op}, +1\rangle \,.\, D)\rangle \text{ (by SECD}_a \text{ rule II3)}\\ &\rightarrow_a \langle k_{m+1}, D\rangle \text{ (by SECD}_a \text{ rule II4).}\end{aligned}$$

Thus $n = 3$, $m = 0$, $V'' = V'$, $D'' = D$ does the job.

rule I5 or I6: This is similar to rule I4 except that in the last step we apply SECD$_a$ rule I5 or I6 respectively.

rule II1: We have $V = (MN)$, $M \rightarrow_{\text{pcf}} M'$, and $V' = (M'N)$. For any D we have

$$\begin{aligned}\langle(MN), D\rangle &\rightarrow_a \langle M, (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle, \text{ and}\\ \langle(M'N), D\rangle &\rightarrow_a \langle M', (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle.\end{aligned}$$

. But by inductive hypothesis, since $M \rightarrow_{\text{pcf}} M'$ we have that there are some m'', n'', V'', D'' such that

$$\begin{aligned}\langle M, (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle &\rightarrow_a^{n''} \langle V'', D''\rangle, \text{ and}\\ \langle M', (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle &\rightarrow_a^{m''} \langle V'', D''\rangle.\end{aligned}$$

So we have

$$\begin{aligned}\langle(MN), D\rangle &\rightarrow_a^{1+n''} \langle V'', D''\rangle, \text{ and}\\ \langle(M'N), D\rangle &\rightarrow_a^{1+m''} \langle V'', D''\rangle,\end{aligned}$$

which covers rewrite rule II1.

rule II2: We have $V = (\supset M)$, $V' = (\supset M')$ with $M \rightarrow_{\text{pcf}} M'$. For any D we have

$$\begin{aligned}
\langle (\supset M), D\rangle &\rightarrow_a \langle \supset, (\langle \mathbf{arg}, M\rangle \,.\, D)\rangle \\
&\rightarrow_a \langle M, (\mathbf{if} \,.\, D)\rangle \text{ (by SECD}_a \text{ rule II2), and similarly} \\
\langle (\supset M'), D\rangle &\rightarrow_a \langle \supset, (\langle \mathbf{arg}, M'\rangle \,.\, D)\rangle \\
&\rightarrow_a \langle M', (\mathbf{if} \,.\, D)\rangle.
\end{aligned}$$

But by inductive hypothesis, since $M \rightarrow_{\text{pcf}} M'$ we have that there are some m'', n'', V'', D'' such that

$$\begin{aligned}
\langle M, (\mathbf{if} \,.\, D)\rangle &\rightarrow_a^{n''} \langle V'', D''\rangle, \text{ and} \\
\langle M', (\mathbf{if} \,.\, D)\rangle &\rightarrow_a^{m''} \langle V'', D''\rangle.
\end{aligned}$$

So we have

$$\begin{aligned}
\langle (\supset M), D\rangle &\rightarrow_a^{1+n''} \langle V'', D''\rangle, \text{ and} \\
\langle (\supset M'), D\rangle &\rightarrow_a^{1+m''} \langle V'', D''\rangle.
\end{aligned}$$

which covers rewrite rule II2.

rule II3: We have $V = (MN)$, $V' = (MN')$, $N \rightarrow_{\text{pcf}} N'$, and M is $+1$, -1, or Z. For any D we have

$$\begin{aligned}
\langle (MN), D\rangle &\rightarrow_a \langle M, (\langle \mathbf{arg}, N\rangle \,.\, D)\rangle \\
&\rightarrow_a \langle N, (\langle \mathbf{op}, M\rangle \,.\, D)\rangle \text{ (by SECD}_a \text{ rule II3), and} \\
\langle (MN'), D\rangle &\rightarrow_a \langle M, (\langle \mathbf{arg}, N'\rangle \,.\, D)\rangle \\
&\rightarrow_a \langle N', (\langle \mathbf{op}, M\rangle \,.\, D)\rangle.
\end{aligned}$$

But by inductive hypothesis, since $N \rightarrow_{\text{pcf}} N'$ we have that there are some m'', n'', V'', D'' such that

$$\langle N, (\langle \mathbf{op}, M\rangle \,.\, D)\rangle \rightarrow_a^{n''} \langle V'', D''\rangle, \text{ and}$$

$$\langle N', (\langle \mathbf{op}, M\rangle \,.\, D)\rangle \quad \rightarrow_a^{m''} \quad \langle V'', D''\rangle.$$

Thus we have

$$\begin{aligned} \langle (MN), D\rangle &\rightarrow_a^{2+n''} \langle V'', D''\rangle, \text{ and} \\ \langle (MN'), D\rangle &\rightarrow_a^{2+m''} \langle V'', D''\rangle. \end{aligned}$$

That covers all the rewrite rules for PCF, thus proving Lemma 1, and thus proving Theorem 1.

Thus we know that our SECD$_a$ machine behaves correctly when given expressions of ground type. If we wanted to talk about the behavior of our SECD$_a$ machine when given expressions of higher type we would want to talk about computational adequacy and full abstractness. One approach to making that work is to assign a "semantics" to the state of the SECD$_a$ machine: Assign to every pair, $\langle V, D\rangle$, an expression M according to the following recursive definition. The function $E(M)$ is

$$\begin{aligned} E(\langle V, ()\rangle) &= V \\ E(\langle V, (\langle \mathbf{arg}, N\rangle, \,.\, D')\rangle) &= E(\langle (V\,N), D'\rangle) \\ E(\langle V, (\mathbf{if} \,.\, D')\rangle) &= E(\langle (\supset V), D'\rangle) \\ E(\langle V, (\langle \mathbf{op}, M\rangle \,.\, D')\rangle) &= E(\langle (MV), D'\rangle) \end{aligned}$$

Having done that we observe that $\rightarrow_a$ is sound in the sense that it preserves the meaning of pairs: I.e. if $\langle V, D\rangle \rightarrow_a \langle V', D'\rangle$ then $\mathcal{A}_A[\![E(\langle V, E\rangle)]\!](\rho) = \mathcal{A}_A[\![E(\langle V', E'\rangle)]\!](\rho)$. (The observation that $\rightarrow_a$ is sound is another case analysis.)

Our Second SECD Machine

The problem with the rewrite rules given by $\rightarrow_a$ is that rule number I3 is not the sort of rule that we expect as a primitive operation for a computer. I.e., we would not really expect to see

the variable substitution operation $[N/\alpha]M$ to be implemented in hardware. We could fix up the rewrite rules given by $\rightarrow_a$ to do one step of an implementation of $[N/\alpha]M$ at a time; That is not what this section does. This section describes a completely different way of dealing with variables. We use *environments* to implement substitution. Using environments is very much in the style of the original SECD machine described in[Lan63], and in the interpreters that are used in MIT's course 6.001, so students may be more familiar with this kind of rewrite rule than the kind we described in the previous section.

Our environments are partial functions from variables to pairs of expressions and environments. I.e. the type of an environment can be thought of as

$$\text{ENV} ::= \text{VAR} \rightarrow \text{ENV}.$$

That notation does not make sense as a mathematical object, but it will turn out to be ok, since we will never have an environment contained in itself. In fact,

Claim 1 *all of the environments that we will use can be built from of a finite number of*

" defined by

$$\rho''(y) = \begin{cases} \langle x, \rho' \rangle & \text{if } y \text{ is the same variable as } x, \\ \rho(y) & \text{otherwise,} \end{cases}$$

is an environment. We denote ρ'' by $\rho[\langle M, \rho' \rangle / x]$.

The reader can check that all of the environments that we use are of this form.

One implementation of environments that we could use would be a list of pairs of variables and pairs of expressions and environments. I.e. an *association list* mapping variables to pairs of expressions and environments. Thus, $\rho_\perp$, the empty environment is represented by the empty list ().

We can represent environments by finite data structures, since all environments are finite. If the list E represents ρ then we can represent $\rho[\langle M, \rho' \rangle / x]$ by the list with first element $\langle x, \langle M, \rho' \rangle \rangle$ and with the rest of the list equal to E. I.e. we represent $\rho[\langle M, \rho' \rangle / x]$ by the list $(\langle x, \langle M, \rho' \rangle \rangle \ . \ E)$. To evaluate $\rho(x)$, all we need to do is look through the list E representing ρ until we find the first pair $\langle y, \langle M, \rho' \rangle \rangle$ in E such that y is the same variable of x. Thus, looking things up in an environment, and creating new environments out of old environment seem to be "primitive enough" to be part of a SECD machine without our having to work too hard to argue that this really is something that corresponds to hardware.

Our second machine takes triples consisting of expressions (the V register), dumps (the D register) and environments (the E register). Our new rewrite rules (which we write as $\rightarrow_b$) will map from such triples to such triples. The V and D registers will serve approximately the same purpose as they did in the SECD_a machine (the D register will have slightly more complex things pushed onto it). The E register will contain an environment which describes how to interpret the free variables of V.

Now we define the function SECD_b in terms of $\rightarrow_b$ in the same way we defined SECD_a in terms of $\rightarrow_a$. We define

$$\text{SECD}_b(M) = c \text{ iff } \langle M, (), \rho_\perp \rangle \rightarrow_b^* \langle c, (), \rho \rangle,$$

where c is a constant in PCF, M is a program in PCF, $()$ is the empty list, and ρ is an environment.

Now we define the relation of $\rightarrow_b$ from state triples to state triples, where $\langle V, D, E \rangle$ contains the V, D, and E registers respectively:

(I1-a) $\langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1, \rho_1\rangle, \langle \mathbf{arg}, M_2, \rho_2\rangle \cdot D'), \rho\rangle \rightarrow_b \langle M_1, \rho_1, D'\rangle$,

(I1-b) $\langle ff, (\mathbf{if}, \langle \mathbf{arg}, M_1, \rho_1\rangle, \langle \mathbf{arg}, M_2, \rho_2\rangle \cdot D'), \rho\rangle \rightarrow_b \langle M_2, \rho_2, D'\rangle$,

(I2) $\langle Y, (\langle \mathbf{arg}, M_1, \rho_1\rangle \cdot D'), \rho\rangle \rightarrow_b \langle (M_1(Y M_1)), D', \rho_1\rangle$,

(I3-a) $\langle (\lambda\alpha M), (\langle \mathbf{arg}, N, \rho_1\rangle \cdot D'), \rho_2\rangle \rightarrow_b \langle M, D', \rho_2[\langle N, \rho_1\rangle/\alpha]\rangle$,

(I3-b) $\langle \alpha, D, \rho_0\rangle \rightarrow_b \langle V_1, D, \rho_1\rangle$ (if $\rho(\alpha) = \langle V_1, \rho_1\rangle$),

(I4) $\langle k_m, (\langle \mathbf{op}, +1\rangle \cdot D'), \rho\rangle \rightarrow_b \langle k_{m+1}, D', \rho\rangle$ (for $m \geq 0$),

(I5) $\langle k_{m+1}, (\langle \mathbf{op}, -1\rangle \cdot D'), \rho\rangle \rightarrow_b \langle k_m, D', \rho\rangle$ (for $m \geq 0$),

(I6-a) $\langle k_0, (\langle \mathbf{op}, Z\rangle \cdot D'), \rho\rangle \rightarrow_b \langle tt, D', \rho\rangle$,

(I6-b) $\langle k_{m+1}, (\langle \mathbf{op}, Z\rangle \cdot D'), \rho\rangle \rightarrow_b \langle ff, D', \rho\rangle$ (for $m \geq 0$),

(II1) $\langle (MN), D, \rho\rangle \rightarrow_b \langle M, (\langle \mathbf{arg}, N, \rho\rangle \cdot D), \rho\rangle$,

(II2) $\langle \supset, (\langle \mathbf{arg}, M_1, \rho_1\rangle \cdot D'), \rho_0\rangle \rightarrow_b \langle M_1, (\mathbf{if} \cdot D'), \rho_1\rangle$,

(II3) $\langle M, (\langle \mathbf{arg}, M_1, \rho_1\rangle \cdot D'), \rho_0\rangle \rightarrow_b \langle M_1, (\langle \mathbf{op}, M\rangle \cdot D'), \rho_1\rangle$ (if M is $(+1)$, (-1), or Z).

We observe that, similarly to $\rightarrow_a$, the $\rightarrow_b$ relation is a partial function from state triples to state triples. We can prove that our new three register SECD machine correctly interprets PCF programs by a theorem which is very similar to Theorem 1.

Theorem 2 *Given any expression E of ground type,* $\mathrm{EVAL}(E)$ *is well defined if and only if* $\mathrm{SECD}_b(E)$ *is well defined, and if they are well defined then* $\mathrm{EVAL}(E) = \mathrm{SECD}_b(E)$.

To prove Theorem 2 we will use the Theorem 1 which gives a relationship between SECD_a and EVAL. To do this we need a way to think of SECD_b state triples as SECD_a state pairs. We do this by means of two functions $\mathcal{E}$ and $\mathcal{D}$ which are defined as follows:

Definition 1 *The function $\mathcal{E}$ maps from pairs of expressions and environments into expressions. We define $\mathcal{E}$ recursively as follows:*

$$\mathcal{E}(\langle \alpha, \rho\rangle) = \begin{cases} \mathcal{E}(\rho(\alpha)) & \text{if } \rho(\alpha) \text{ is defined,} \\ \alpha & \text{if } \rho(\alpha) \text{ is not defined,} \end{cases}$$

$$\mathcal{E}(\langle c, \rho\rangle) = c \text{ if } c \text{ is a constant,}$$

$$\mathcal{E}(\langle (MN), \rho\rangle) = (\mathcal{E}(\langle M, \rho\rangle)\, \mathcal{E}(\langle N, \rho\rangle)),$$

$$\mathcal{E}(\langle(\lambda\alpha M),\rho\rangle) \;=\; (\lambda\alpha'\mathcal{E}(\langle[\alpha'/\alpha]M,\rho\rangle)) \quad \textit{where } \alpha' \textit{ is a variable such that } \rho(\alpha') \textit{ is not defined and } \alpha' \textit{ does not appear in } M.$$

Definition 2 *The function $\mathcal{D}$ maps from stack values for the* SECD$_a$ *machine to stack values for the* SECD$_b$ *machine. We deifne $\mathcal{D}$ recursively as follows:*

$$\mathcal{D}(()) \;=\; () \textit{ (the empty stack maps to the empty stack),}$$
$$\mathcal{D}((\langle \mathbf{arg}, M, \rho\rangle \,.\, D)) \;=\; (\langle \mathbf{arg}, \mathcal{E}(\langle M,\rho\rangle)\rangle \,.\, \mathcal{D}(D)).$$

Then we show the relationship between SECD$_b$ state triples and SECD$_a$ state pairs in the following lemma:

Lemma 2 *1. For every closed expression M (i.e. M has no free variables), and for every non-negative integer n, if*

$$\langle M, (), \rho_\perp\rangle \rightarrow_b^n \langle M', D', \rho'\rangle,$$

then there is a non-negative integer m such that

$$\langle M, ()\rangle \rightarrow_a^m \langle \mathcal{E}(\langle M', \rho'\rangle), \mathcal{D}(D')\rangle.$$

2. For every closed expression M and every integer m, if

$$\langle M, ()\rangle \rightarrow_a^m \langle M'', D''\rangle,$$

then there is an n such that

$$\langle M, (), \rho_\perp\rangle \rightarrow_b^n \langle M', D', \rho'\rangle,$$

and $M'' = \mathcal{E}(\langle M', \rho'\rangle)$ and $D'' = \mathcal{D}(D')$.

The proof for Lemma 2 is postponed for a moment while we use Lemma 2 to prove Theorem 2. We first need another lemma which tells us the conditions under which $\mathcal{E}(\langle M,\rho\rangle)$ can be a constant:

Lemma 3 *If* $\mathcal{E}(\langle M, \rho\rangle) = c$ *for a constant* c *then* $\langle M, (), \rho\rangle \rightarrow_b^* \langle c, (), \rho'\rangle$ *for some* ρ'.

The reason that Lemma 2 proves Theorem 2 is that if we know that $\mathrm{EVAL}(M) = c$ for a constant, then we know that $\mathrm{SECD}_a(M) = c$. I.e. $\langle M, ()\rangle \rightarrow_a^m \langle c, ()\rangle$ for some m. Thus there is an n such that $\langle M, (), \rho_\perp\rangle \rightarrow_b^n \langle M', D', \rho'\rangle$ and $\mathcal{E}(\langle M', \rho'\rangle) = c$ and $\mathcal{D}(D) = ()$. This means that $D = ()$ since the only stack that $\mathcal{D}$ takes to the empty stack is the empty stack. If we know that $\mathcal{E}(\langle M', \rho'\rangle) = c$, by Lemma 3 we have that $\langle M, (), \rho_\perp\rangle \rightarrow_b^* \langle c, (), \rho''\rangle$ for some ρ'', in which case $\mathrm{SECD}_b(M) = c$.

Proof of Lemma 3: According to Claim 1, the set of all environments that we use has a lot of structure. There is a partial ordering on those environments induced by the transitive closure of "immediately smaller", where we say that ρ is "immediately smaller" than ρ' if for some variable α and some expression M, $\rho'(\alpha) = \langle M, \rho\rangle$. Thus ρ is "smaller" than ρ' if it ρ is "mentioned" somewhere inside ρ'. This partial ordering has the property that there are no infinite decreasing sequences of environments $(\rho_1, \rho_2, \ldots)$ with ρ_{i+1} smaller than ρ_i for all $i \geq 1$. Thus we can do a proof by induction on environments using this structure.

Note that if M is of the form $(N_1 N_2)$ or $(\lambda \alpha N)$ then there is no way that $\mathcal{E}(\langle M, \rho\rangle)$ could be equal to c. Thus we can assume that M is either a constant or a variable. If M is a constant c then $\mathcal{E}(\langle M, \rho\rangle) = c$ so we have $\langle M, (), \rho\rangle \rightarrow_b^0 \langle c, (), \rho\rangle$ so the lemma is true. We will assume from here that M is a variable. We know that $\rho(M)$ must be well defined, or else we would have $\mathcal{E}(M, \rho) = M$ which is not a constant. Thus $\rho(M) = \langle M', \rho'\rangle$ where ρ' is smaller than ρ. But

$$\mathcal{E}(\langle M, \rho\rangle) = \mathcal{E}(\rho(M)) = \mathcal{E}(\langle M', \rho'\rangle) = c,$$

and so by inductive hypothesis,

$$\langle M', (), \rho'\rangle \rightarrow_b^* \langle c, (), \rho''\rangle$$

for some ρ''. But note that

$$\langle M, (), \rho\rangle \rightarrow_b \langle M', (), \rho'\rangle,$$

so we have

$$\langle M, (), \rho \rangle \rightarrow_b^* \langle c, (), \rho'' \rangle$$

for some ρ'', which proves Lemma 3.

To prove Lemma 2 we need a lemma to tell us something about how $\mathcal{E}$ and $\mathcal{D}$ interacts with $\rightarrow_b$, and a lemma that tells us how $\mathcal{E}$ interacts with variable substitution in expressions and environments.

Lemma 4 *Given expressions N and N', and a variable α, and and environments ρ and ρ', and a variable α' not defined by ρ or ρ' and not appearing in N or N', we have*

$$\mathcal{E}(\langle N, \rho[\langle N', \rho' \rangle / \alpha] \rangle) = [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle [\alpha'/\alpha]N, \rho \rangle).$$

Proof of Lemma 4: This proof is by structural induction on N:

- The first base case is for when N is a constant. In this case $\mathcal{E}(\langle c, \rho[\langle N', \rho' \rangle / \alpha] \rangle) = c$ and

$$\begin{aligned} [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle [\alpha'/\alpha]c, \rho \rangle) &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle c, \rho \rangle) \\ &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']c \\ &= c. \end{aligned}$$

- Another base case is when N is the variable α. In this case

$$\begin{aligned} [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle [\alpha'/\alpha]\alpha, \rho \rangle) &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle \alpha', \rho \rangle) \\ &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\alpha' \text{ (since } \rho(\alpha') \text{ is undefined)}, \\ &= \mathcal{E}(\langle N', \rho' \rangle) \\ &= \mathcal{E}(\alpha, \rho[\langle N', \rho' \rangle / \alpha]). \end{aligned}$$

- Another base case is when $N = \beta$ is a different variable from α and $\rho(\beta)$ is defined. We have

$$[\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle [\alpha'/\alpha]\beta, \rho \rangle) = [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle \beta, \rho \rangle)$$

$$
\begin{aligned}
&= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\rho(\beta)) \\
&= \mathcal{E}(\rho(\beta)) \text{ (since } \alpha' \text{ is new)}, \\
&= \mathcal{E}(\beta, \rho[\langle N', \rho' \rangle/\alpha]).
\end{aligned}
$$

- The last base case is when $N = \beta$ a different variable from α and $\rho(\beta)$ is not defined. We have

$$
\begin{aligned}
[\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle [\alpha'/\alpha]\beta, \rho \rangle) &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\mathcal{E}(\langle \beta, \rho \rangle) \\
&= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\beta \\
&= \beta \\
&= \mathcal{E}(\beta, \rho[\langle N', \rho' \rangle/\alpha]).
\end{aligned}
$$

- The first inductive case is when $N = (M_1, M_2)$: In this case we have (using $\{\ldots\}$ to parenthesize)

$$
\begin{aligned}
[\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle [\alpha'/\alpha](M_1 M_2), \rho \rangle)\} &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle ([\alpha'/\alpha]M_1\ [\alpha'/\alpha]M_2), \rho \rangle)\} \\
&= ([\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle [\alpha'/\alpha]M_1, \rho \rangle)\} \\
&\quad\ [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle [\alpha'/\alpha]M_2, \rho \rangle)\}) \\
&= (\mathcal{E}(\langle M_1, \rho[\mathcal{E}(\langle N', \rho' \rangle)/\alpha'] \rangle) \\
&\quad\ \mathcal{E}(\langle M_2, \rho[\mathcal{E}(\langle N', \rho' \rangle)/\alpha'] \rangle)) \\
&\quad\ \text{(by inductive hypothesis)} \\
&= \mathcal{E}(\langle (M_1 M_2), \rho[\mathcal{E}(\langle N', \rho' \rangle)/\alpha'] \rangle).
\end{aligned}
$$

- The next inductive case is when N is of the form $\lambda\alpha M$. We have

$$
\begin{aligned}
[\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle [\alpha'/\alpha](\lambda\alpha M), \rho \rangle)\} &= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha']\{\mathcal{E}(\langle (\lambda\alpha M), \rho \rangle)\} \\
&= [\mathcal{E}(\langle N', \rho' \rangle)/\alpha'](\lambda\alpha''\mathcal{E}(\langle [\alpha''/\alpha]M, \rho \rangle)) \\
&= (\lambda\alpha''\mathcal{E}(\langle [\alpha''/\alpha]M, \rho \rangle)) \text{ (since } \alpha' \text{ is new)}
\end{aligned}
$$

$$
\begin{aligned}
&= \mathcal{E}(\langle(\lambda\alpha M), \rho\rangle) \text{ (def'n of } \mathcal{E}) \\
&= \mathcal{E}(\langle(\lambda\alpha M, \rho[\langle N', \rho'\rangle/\alpha])\rangle).
\end{aligned}
$$

- The last inductive case is when N is of the form $\lambda\beta M$. (I.e. the variable β is not the same as the variable α.) In this case we have

$$
\begin{aligned}
[\mathcal{E}(\langle N', \rho'\rangle)/\alpha']\{\mathcal{E}(\langle[\alpha'/\alpha](\lambda\beta M), \rho\rangle)\} &= [\mathcal{E}(\langle N', \rho'\rangle)/\alpha']\{\mathcal{E}(\langle(\lambda\beta[\alpha'/\alpha]M), \rho\rangle)\} \\
&= [\mathcal{E}(\langle N', \rho'\rangle)/\alpha']\{\mathcal{E}(\langle(\lambda\beta[\alpha'/\alpha]M), \rho\rangle)\} \\
&= [\mathcal{E}(\langle N', \rho'\rangle)/\alpha'](\lambda\beta'\mathcal{E}(\langle[\beta'/\beta][\alpha'/\alpha]M, \rho\rangle)) \\
&= [\mathcal{E}(\langle N', \rho'\rangle)/\alpha'](\lambda\beta'\mathcal{E}(\langle[\alpha'/\alpha][\beta'/\beta]M, \rho\rangle)) \\
&= (\lambda\beta''[\mathcal{E}(\langle N', \rho'\rangle)/\alpha']\{\mathcal{E}(\langle[\alpha'/\alpha][\beta'/\beta]M, \rho\rangle)\}) \\
&= (\lambda\beta''\mathcal{E}(\langle[\beta'/beta]M, \rho[\langle N', \rho'\rangle/\alpha]\rangle)) \\
&= \mathcal{E}(\langle(\lambda\beta M), \rho[\langle N', \rho'\rangle/\alpha]\rangle)
\end{aligned}
$$

  which finishes the case analysis.

Lemma 5 *If M is a closed term and*

$$\langle M, (), \rho_\perp\rangle \rightarrow_b^n \langle M', D', \rho'\rangle,$$

then $\mathcal{E}(\langle M', \rho'\rangle)$ is closed and if $D' = (d_1, d_2, \ldots, d_k)$ (i.e. D' has k elements in it) then $\mathcal{D}(D') = (e_1, \ldots, e_k)$ for some $e_1, \ldots, e_k$ and

- *if d_i is of the form $\langle \mathbf{arg}, M_i, \rho_i\rangle$ then $e_1 = \langle \mathbf{arg}, \mathcal{E}(\langle M_i, \rho_i\rangle)\rangle$ is closed,*
- *if d_i is of the form $\langle \mathbf{op}, M\rangle$ or* **if** *then $e_i = d_i$.*

Proof of Lemma 5: The interesting part of this proof is that $\mathcal{E}(M', \rho')$ is closed and that for the d_i's of the form $\langle \mathbf{arg}, M_i, \rho_i\rangle$ that $\mathcal{E}(\langle M_i, \rho_i\rangle)$ is closed, since the rest follows directly from the definition of $\mathcal{E}$ and $\mathcal{D}$. We prove the interesting part by induction on n and leave the rest of the proof to the reader.

- For $n = 0$, Lemma 5 is just the statement that if M is closed then $\mathcal{E}(\langle M, \rho_\perp \rangle)$ is closed, since $D' = ()$. But $\mathcal{E}(\langle M, \rho_\perp \rangle)$ is closed since the only thing that $\mathcal{E}$ does when its environment argument is $\rho_\perp$ is rename bound variables.

- For $n > 0$, we assume that Lemma 5 is true for $n - 1$, i.e. when we have

$$\langle M, (), \rho \rangle \rightarrow_b^* \langle M', D', \rho' \rangle$$

we know that $\mathcal{E}(\langle M', \rho' \rangle)$ is closed and that the entries of $\mathcal{D}(D')$ which are of the form $\langle \mathbf{arg}, M_i \rangle$ have M_i closed. There are two cases. If $\langle M', D', \rho' \rangle$ does not transform to anything under $\rightarrow_b$ then we are done. Otherwise we have

$$\langle M', D', \rho' \rangle \rightarrow_b \langle M'', D'', \rho'' \rangle$$

for some M'', D'', and ρ''. We do a case analysis on which rule it was that made this transition happen.

  - Most of the cases are fairly simple. We do the example of when rule I1-a was the one that made the nth transformation happen. In this case we have

$$\langle M', D', \rho' \rangle = \langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1, \rho_1 \rangle, \langle \mathbf{arg}, M_2, \rho_2 \rangle \ . \ D'''), \rho' \rangle.$$

    We know that $\mathcal{E}(\langle M_1, \rho_1 \rangle)$ closed and the entries of $\mathcal{D}(D''')$'s which are of the form $\langle \mathbf{arg}, N_i \rangle$ all have closed N_i by inductive hypothesis. But in this case

$$\begin{aligned} \langle M', D', \rho' \rangle &= \langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1, \rho_1 \rangle, \langle \mathbf{arg}, M_2, \rho_2 \rangle \ . \ D'''), \rho' \rangle \\ &\rightarrow_b \langle M_1, D''', \rho_1 \rangle \end{aligned}$$

    which shows that rule I1-a of $\rightarrow_b$ preserves the properties of Lemma 5.

  - The one other case that calls for proof is rule I3-a. In this case we have

$$\langle (\lambda \alpha M_1), (\langle \mathbf{arg}, M_2, \rho_2 \rangle \ . \ D'), \rho_1 \rangle \rightarrow_b \langle M_1, D', \rho_1[\langle M_2, \rho_2 \rangle / \alpha] \rangle.$$

    But by Lemma 4 we have

$$\mathcal{E}(\langle M_1, \rho_1[\langle M_2, \rho_2 \rangle / \alpha] \rangle) = [\mathcal{E}(\langle M_2, \rho_2 \rangle) / \alpha'] \mathcal{E}(\langle [\alpha' / \alpha] M_1, \rho_1 \rangle)$$

but

$$\mathcal{E}(\langle(\lambda\alpha M_1), \rho_1\rangle) = (\lambda\alpha' \mathcal{E}(\langle[\alpha'/\alpha]M, \rho_2\rangle))$$

is closed by inductive hypothesis, so $\mathcal{E}(\langle[\alpha'/\alpha]M, \rho_2\rangle)$ has no free variables except for possibly α', and we know by inductive hypothesis that $\mathcal{E}(\langle N_1, \rho_1\rangle)$ is closed, so if we replace all free occurences of α' in $\mathcal{E}(\langle[\alpha'/\alpha]M, \rho_2\rangle)$ by $\mathcal{E}(\langle N_1, \rho_1\rangle)$ then we get a closed term and we conclude that rule I3-a preserves the properties of Lemma 5.

Thus proving Lemma 5.

Now we prove Lemma 2 using Lemma 4 and Lemma 5. This proof is by induction: If we have

$$\langle M, (), \rho_\perp\rangle \rightarrow_b^n \langle M', D', \rho'\rangle,$$

and by induction

$$\langle M, ()\rangle \rightarrow_a^m \langle \mathcal{E}(\langle M', \rho'\rangle), \mathcal{D}(D')\rangle,$$

then we consider the next transformation under $\rightarrow_b$.

If there is no such transformation, then we argue that there is no next transformation of $\rightarrow_b$ by looking at the the expressions which could end up not being transformed and how they are acted on by $\mathcal{E}$. Since there is no next transformation of $\rightarrow_b$, we must have $\mathcal{E}(\langle M', \rho'\rangle) = c$, a constant (by Theorem 1). But by Lemma 3 that means that $\langle M', D', \rho'\rangle$ must go to $\langle c, (), \rho''\rangle$ for some ρ'', but since there is no next transformation, we must have $\mathrm{SECD}_b(M) = c$ which is what we wanted.

If there is a next transformation, then we do a case analysis on which rule is used to make the transformation

$$\langle M', D', \rho'\rangle \rightarrow_b \langle M'', D'', \rho''\rangle.$$

And then we show that there is some $m' \leq 1$ such that

$$\langle \mathcal{E}(\langle M', \rho'\rangle), \mathcal{D}(D')\rangle \rightarrow_a^{m'} \langle \mathcal{E}(\langle M'', \rho''\rangle), \mathcal{D}(D'')\rangle.$$

If we show that such m' exists and is less than or equal to one, then part 2 of Lemma 2 is true since there will be no way to "skip" m's without "skipping" n's.

Our case analysis of the rule which is used is as follows:

rule (I1-a) We have

$$\begin{aligned}
\langle M, (), \rho_\perp \rangle &\rightarrow_b^n \langle tt, (\mathbf{if}, \langle \mathbf{arg}, M_1, \rho_1 \rangle, \langle \mathbf{arg}, M_2, \rho_2 \rangle \,.\, D'), \rho \rangle \\
&\rightarrow_b \langle M_1, D', \rho_1 \rangle, \text{ and} \\
\langle M, () \rangle &\rightarrow_a^m \langle tt, (\mathbf{if}, \langle \mathbf{arg}, \mathcal{E}(\langle M_1, \rho_1 \rangle) \rangle, \langle \mathbf{arg}, \mathcal{E}(\langle M_2, \rho_2 \rangle) \rangle \,.\, \mathcal{D}(D')) \rangle \\
&\rightarrow_a \langle \mathcal{E}(\langle M_1, \rho_1 \rangle), \mathcal{D}(D') \rangle.
\end{aligned}$$

So the number of extra steps is exactly one and things match up.

Any other rule except for II1, I3-a, or I3-b: These are very similar to the case for rule I1-a, and are left to the reader.

rule II1: In this case we have

$$\begin{aligned}
\langle M, (), \rho_\perp \rangle &\rightarrow_b^n \langle (M_1 M_2), D, \rho \rangle \\
&\rightarrow_b \langle M_1, (\langle \mathbf{arg}, M_2, \rho \rangle \,.\, D), \rho \rangle, \text{ and} \\
\langle M, () \rangle &\rightarrow_a^m \langle \mathcal{E}(\langle (M_1 M_2), \rho \rangle), \mathcal{D}(D) \rangle \\
&= \langle (\mathcal{E}(\langle M_1, \rho \rangle)\ \mathcal{E}(\langle M_2, \rho \rangle)), \mathcal{D}(D) \rangle \\
&\rightarrow_a \langle \mathcal{E}(M_1, \rho), (\langle \mathbf{arg}, \mathcal{E}(\langle M_2, \rho \rangle) \rangle \,.\, \mathcal{D}(D)) \rangle.
\end{aligned}$$

So the number of steps is exactly one to match things up.

rule I3-a: We have

$$\begin{aligned}
\langle M, (), \rho_\perp \rangle &\rightarrow_b^n \langle (\lambda \alpha M_1), (\langle \mathbf{arg}, M_2, \rho_2 \rangle \,.\, D), \rho_1 \rangle \\
&\rightarrow_b \langle M_1, D, \rho_2 [\langle M_2, \rho_2 \rangle / \alpha] \rangle, \text{ and} \\
\langle M, () \rangle &\rightarrow_a^m \langle \mathcal{E}(\langle (\lambda \alpha M_1), \rho_1 \rangle), (\langle \mathbf{arg}, \mathcal{E}(\langle M_2, \rho_2 \rangle) \rangle \,.\, \mathcal{D}(D)) \rangle
\end{aligned}$$

$$
\begin{aligned}
&= \langle(\lambda\alpha'\mathcal{E}(\langle[\alpha'/\alpha]M_1,\rho_1\rangle)),(\langle \mathbf{arg}, \mathcal{E}(\langle M_2,\rho_2\rangle)\rangle \cdot \mathcal{D}(D))\rangle \\
\rightarrow_a\ &\langle[\mathcal{E}(\langle M_2,\rho_2\rangle)/\alpha']\mathcal{E}(\langle[\alpha'/\alpha]M_1,\rho_1\rangle),\mathcal{D}(D)\rangle \\
&= \langle\mathcal{E}(\langle M_2,\rho_2[\langle M_1,\rho_1\rangle/\alpha]\rangle),\mathcal{D}(D)\rangle \text{ (by Lemma 4).}
\end{aligned}
$$

So things match up after one step.

rule I3-b: We have

$$
\begin{aligned}
\langle M,(),\rho_\perp\rangle\ &\rightarrow_b^n\ \langle\alpha,D,\rho\rangle \\
&\rightarrow_a\ \langle M_2,D,\rho_2\rangle \text{ (where } \rho(\alpha)=\langle M_2,\rho_2\rangle\text{), and} \\
\langle M,()\rangle\ &\rightarrow_a^m\ \langle\mathcal{E}(\langle\alpha,\rho\rangle),\mathcal{D}(D)\rangle \\
&=\ \langle\mathcal{E}(\langle\rho(\alpha)\rangle),\mathcal{D}(D)\rangle.
\end{aligned}
$$

So everything matches up after no more transformations of $\rightarrow_a$. I.e. there is an $\rightarrow_b$ transformation which is just "part of" an $\rightarrow_a$ transformation. Thus we have Lemma 2.

Conclusion

We have shown two SECD style machines for interpreting PCF. The first machine, SECD_a, is in some sense closer to the original $\rightarrow_{\text{pcf}}$ rewrite rules and we proved that $\text{SECD}_a(M) = EVAL(M)$. The second machine, SECD_b, is in some sense more the sort of thing we expect from a SECD machine, and we proved that $\text{SECD}_b(M) = \text{SECD}_a(M)$. Readers who are familiar with the SECD machines in the litarture should feel more familiar with the second SECD machine than with the first since most of th SECD machines in the literature use environments in a way similar to the way we use environments. As a last note, we have implemented both of these SECD machines in Common Lisp, and the SECD_b machine is orders of magnitude better (in terms of speed and the amount of consing) than the SECD_a machine in our implementation. We believe that this reflects something fundamental about the different machines rather than that we might have done a better job implementing the

$SECD_b$ machine than the $SECD_a$ machine. Of course if one were really interested in getting even better performance out of a practical implementation of PCF, one might try to do away with the variables *per se* at run time and compile the PCF code into something which can be run even faster (e.g. combinator code).

References

[Lan63] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1963.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 1:125–159, 1977.